Technical Report
CMU/SEI-93-TR-3
ESC-TR-93-180

Carnegie-Mellon University
Software Engineering Institute

Software Architecture for Shared Information Systems

Mary Shaw

March 1993

Technical Report
CMU/SEI-93-TR-3
ESC-TR-93-180
March 1993

Software Architecture for Shared Information Systems

Mary Shaw

School of Computer Science
and Software Engineering Institute

Software Engineering Information Modeling Project

This report will also appear as Carnegie Mellon University
School of Computer Science Technical Report No. CMU-CS-93-126.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department Defense.
This report was funded by the U.S. Department of Defense.

The work reported here was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense), by a grant from Siemens Corporate Research, and by the Department of Defense Advanced Research Project Agency under grant MDA972-92-J-1002. The views and conclusions are those of the author.

Table of Contents

1. Introduction **1**
- 1.1 Systems Integration 2
- 1.2 Shared Information Systems 3
- 1.3 Design Levels 4
- 1.4 External Software Systems 6

2. Database Integration **9**
- 2.1. Batch Sequential 9
- 2.2. Simple Repository 12
- 2.3. Virtual Repository 16
- 2.4. Hierarchical Layers 18
- 2.5. Evolution of Shared Information Systems in Business Data Processing 20

3. Integration in Software Development Environments **23**
- 3.1. Batch Sequential 23
- 3.2. Transition from Batch Sequential to Repository 23
- 3.3. Repository 26
- 3.4. Hierarchical Layers 27
- 3.5. Evolution of Shared Information Systems in Software Development Environments 29

4. Integration in Building Design **31**
- 4.1. Repository 32
- 4.2. *Intelligent Control* 34
- 4.3. Evolution of Shared Information Systems in Building Design 36

5. Architectural Structures for Shared Information Systems **37**
- 5.1 Variants on Data Flow Systems 37
- 5.2. Variants on Repositories 38

6. Conclusions **41**

Acknowledgments 43

References 45

Software Architectures for Shared Information Systems[1]

Abstract: Software system design takes place at many levels. Different kinds of design elements, notations, and analyses distinguish these levels. At the *software architecture* level, designers combine subsystems into complete systems. This paper studies some of the common patterns, or idioms, that guide these configurations. Results from software architecture offer some insight into the problems of systems integration—the task of connecting individual, isolated, pre-existing software systems to provide coherent, distributed solutions to large problems. As computing has become more sophisticated, so too have the software structures used in the integration task. This paper reviews historical examples of shared information systems in three different applications whose requirements share some common features about collecting, manipulating, and preserving large bodies of complex information. These applications have similar architectural histories in which a succession of designs responds to new technologies and new requirements for flexible, highly dynamic responses. A common pattern, the *shared information system evolution pattern,* appears in all three areas.

1. Introduction

Software system design takes place at many levels, each with its own concerns. We learn from computer hardware design that each of these levels has its own elements and composition operators and its own notations, analysis tools, and design rules. From the 1960s through the 1980s software developers concentrated on the programming level. At this level, so-called higher-level programming languages provide for the definitions of algorithms and data structures using the familiar programming language control statements, types, and procedures. Now we are turning our attention to the architectural level, in which patterns for organizing module-scale components guide software system design.

---

[1]To appear in *Mind Matters: Contributions to Cognitive and Computer Science in Honor of Allen Newell.* David Steier and Tom Mitchell (eds.), Hillsdale, N.J.: Lawrence Erlbaum Associates (to appear).

1.1 Systems Integration

Large software systems are often integrated from pre-existing systems. The designer of such a system must accommodate very different—often incompatible—conceptual models, representations, and protocols in order to create a coherent synthesis. Systems integration is a problem-solving activity that entails harnessing and coordinating the power and capabilities of information technology to meet a customer's needs. It develops megasystems in which pre-existing independent systems become subsystems—components that must interact with other components. Successful integration requires solution of both organizational and technical problems:

- understanding the current organizational capabilities and processes
- re-engineering and simplification of processes with a system view
- standardizing on common data languages and system architectures
- automation of processes and systems

Five kinds of issues motivate companies to invest in systems integration (CSTB 1992, pp. 16-21):

- For many organizations, experiences with information technology have not lived up to expectations.
- The proliferation of information technology products and vendors has produced the need for connectivity and interoperability.
- An installed base of information technology has to accommodate new technology and new capabilities.
- Advances in technology, combined with growing appreciation of what can be accomplished with that technology, have prompted firms to search for new applications and sources of competitive advantage.
- In an increasingly global economy, firms must rely on telecommunications and information technology to manage and coordinate their operations and to stay abreast of international competitors.

Corporate mergers and reorganizations, in particular, create needs for compatibility among systems developed under different assumptions about representation and interaction. The task is difficult: it involves large, untidy problems; incomplete, imprecise, and inconsistent requirements; and "legacy" systems that must be retained rather than replaced. For systems integration to be useful, it must be globally effective within the organization. The focus of this paper is on the enabling technologies rather than the organizational questions.

The essential enabling technologies are of several kinds (CSTB 1992, Nilsson et al 1990):

- *Architecture:* System organization; kinds of components, kinds of interactions, and patterns of overall organization; ability to evolve; consistency with available modular building blocks for hardware, software, and databases; standardization and open systems
- *Semantics:* Representations; conceptual consistency; semantic models; means for handing inconsistencies
- *Connectivity:* Mechanisms for moving data between systems and initiating action in other systems; communication platforms with flexible linkages or interfaces; network management and reliability; security
- *Interaction:* Granularity; user interfaces; interoperability; simplicity; underlying consistency of presentation

The technologies for architecture are of primary interest here; to a certain extent these are inseparable from semantics.

1.2 Shared Information Systems

One particularly significant class of large systems is responsible for collecting, manipulating, and preserving large bodies of complex information. These are *shared information systems.* Systems of this kind appear in many different domains; this paper examines three. The earliest shared information systems consisted of separate programs for separate subtasks. Later, multiple independent processing steps were composed into larger tasks by passing data in a known, fixed format from one step to another. This organization is not flexible in responding to variations or discrepancies in data. Nor is it tolerant of structural modification, especially the addition of components developed under different assumptions. It is also not responsive to the needs of interactive processing, which must handle individual requests as they arrive.

Still later, often when requirements for interaction appear, new organizations allowed independent processing subsystems to interact through a shared data store. While this organization is an improvement, it still encounters integration problems—especially when multiple data stores with different representations must be shared, when the system is distributed, when many user tasks must be served, and when the suite of processing and data subsystems changes regularly. Several newer approaches now compensate for these differences in representation and operating assumptions, but the problem is not completely solved. A common pattern, the *shared information system evolution pattern*, is evident in the application areas examined here.

1.3 Design Levels

System design takes place at many levels. It is useful to make precise distinctions among those levels, for each level appropriately deals with different design concerns. At each level we find *components*, both primitive and composite; *rules of composition* that allow the construction of nonprimitive components, or systems; and *rules of behavior* that provide semantics for the system (Bell and Newell 1971, Newell 1982, Newell 1990). Since these differ from one level to another, we also find different notations, design problems, and analysis techniques at each level. As a result, design at one level can proceed substantially autonomously of any other level. But levels are also related, in that elements at the base of each level correspond to—are implemented by—constructs of the level below.

The hierarchy of levels for computer hardware systems is familiar and appears in Figure 1 (Bell and Newell 1971, p. 3). Note first that each level deals with different content. Different kinds of structures guide design with different sets of components. Different notations, analysis techniques, and design issues accompany the differences of content matter. Note also that each level admits of substructure: abstraction and composition take place within each level, in terms of the components and structures of that level. In addition, there is an established transformation from the primitive components at the bottom of each level to (probably nonprimitive) components of the level below.

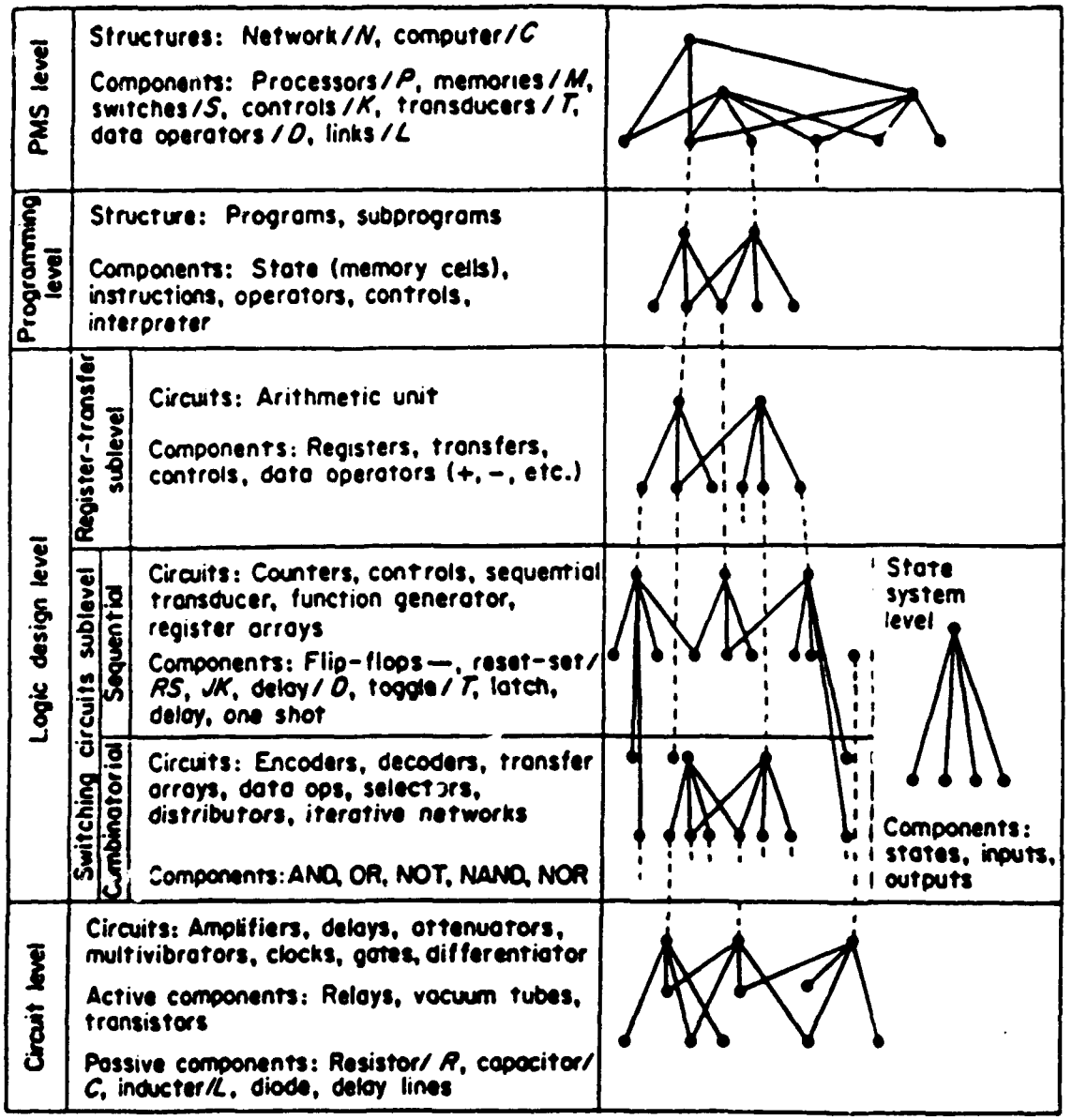

Figure 1: Computer hardware design levels

Software, too, has its design levels. We can identify at least

- *Architecture*, where the design issues involve overall association of system capability with components; components are modules, and interconnections among modules are handled in a variety of ways, all of which seem to be expressed as explicit sets of procedure calls.
- *Code*, where the design issues involve algorithms and data structures; the components are programming language primitives such as numbers, characters, pointers, and threads of control; primitive operators are the arithmetic and data manipulation primitives of the language; composition mechanisms include records, arrays, and procedures.
- *Executable*, where the design issues involve memory maps, data layouts, call stacks, and register allocations; the components are bit patterns supported by hardware, and the operations and compositions are described in machine code.

These roughly track the higher levels of hardware design. The executable and code levels for software are well understood. However, the architecture level is currently understood mostly at the level of intuition, anecdote, and folklore. It is common for a description of a software system to include a few paragraphs of text and a box-and-line diagram, but there is neither uniform syntax nor uniform semantics for interpreting the prose and the diagrams. Our concern here is in improving understanding and precision at the software architecture level. At this level the components are programs, modules, or systems; a rich collection of interchange representations and protocols connect the components; and well-known system patterns guide the compositions (Garlan and Shaw 1993).

1.4 External Software Systems

Recent work on intelligent integration of *external software systems* offers some hope for improving the sophistication of our integration techniques. Newell and Steier (1991) suggest that the work on agent-ESS systems may contribute to software engineering by making the power of computer software more easily accessible in the service of computational tasks. An intelligent system would learn to recognize aberrations when they arise and compensate for them, and it would adapt to new protocols and representations when the suite of available components changes.

This paper explores what happens when independent systems become components of larger systems. It examines three examples of shared information systems:

- *Data processing,* driven primarily by the need to build business decision systems from conventional databases
- *Software development environments,* driven primarily by the need to represent and manipulate programs and their designs.
- *Building design,* driven primarily by the need to couple independent design tools to allow for the interactions of their results in structural design

We will see how the software architectures of these systems changed as technology and demands on system performance changed. We close by surveying the architectural constructs used to describe the examples and examining the prospects for intelligent integration.

2. Database Integration

Business data processing has traditionally been dominated by database management, in particular by database updates. Originally, separate databases served separate purposes, and implementation issues revolved around efficient ways to do massive coordinated periodic updates. As time passed, interactive demands required individual transactions to complete in real time. Still later, as databases proliferated and organizations merged, information proved to have value far beyond its original needs. Diversity in representations and interfaces arose, information began to appear redundantly in multiple databases, and geographic distribution added communication complexity. As this happened, the challenges shifted from individual transaction processing to integration.

Individual database systems must support transactions of predetermined types and periodic summary reports. Bad requests require a great deal of special handling. Originally the updates and summary reports were collected into batches, with database updates and reports produced during periodic batch runs. However, when interactive queries became technologically possible, the demand for interaction made generated demand for on-line processing of both individual requests and exceptions. Reports remained on roughly the same cycles as before, so reporting became decoupled from transaction processing.

As databases became more common, information about a business became distributed among multiple databases. This offered new opportunities for the data to become inconsistent and incomplete. In addition, the representations, or schemas, for different databases were usually different; even the portion of the data shared by two databases is likely to have representations in each database. The total volume of data to handle is correspondingly larger, and it is often distributed across multiple machines. Two general strategies emerged for dealing with data diversity: unified schemas and multi-databases.

2.1. Batch Sequential

Some of the earliest large computer applications were databases. In these applications individual database operations—transactions—were collected into large batches. The application consisted of a small number of large standalone programs that performed sequential updates on flat (unstructured) files. A typical organization included:

- a massive *edit program*, which accepted transaction inputs and performed such validation as was possible without access to the database

- a massive *transaction sort*, which got the transactions into the same order as the records on the sequential master file
- a sequence of *update programs*, one for each master file; these huge programs actually executed the transactions by moving sequentially through the master file, matching each type of transaction to its corresponding account and updating the account records
- a *print program* that produced periodic reports

The steps were independent of each other; they had to run in a fixed sequence; each ran to completion, producing an output file in a new format, before the next step began. This is a *batch sequential* architecture. The organization of a typical batch sequential update system appears in Figure 2 (Best 1990, p. 29). This figure also shows the possibility of on-line queries (but not modifications). In this structure the files to support the queries are reloaded periodically, so recent transactions (e.g., within the past few days) are not reflected in the query responses.

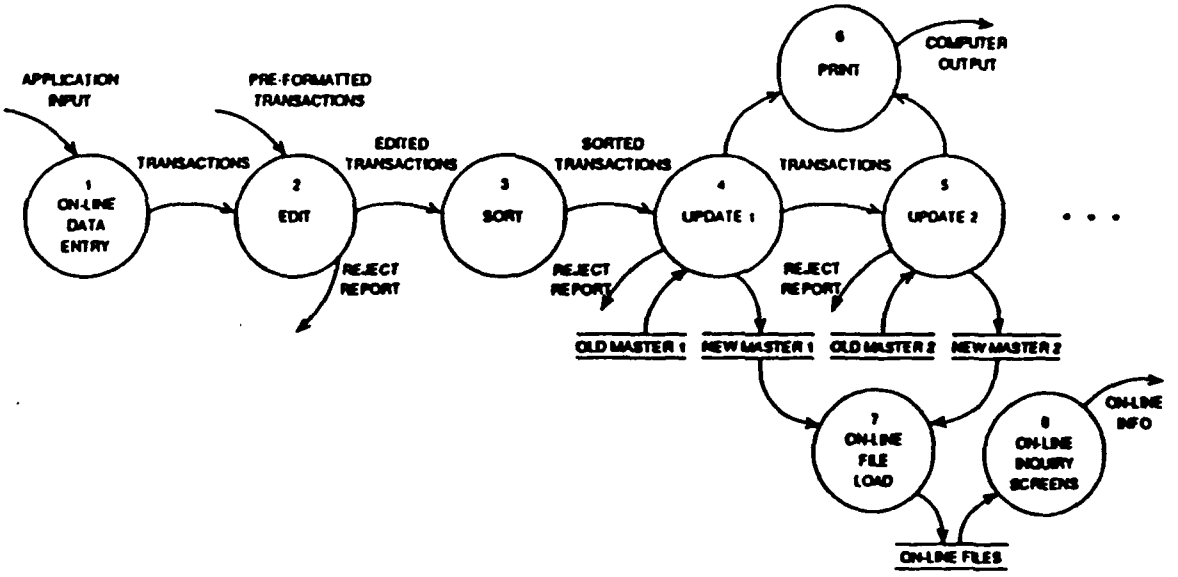

Figure 2: Data flow diagram for batch databases

Figure 2 is a Yourdon data flow diagram. Processes are depicted as circles, or "bubbles"; data flow (here, large files) is depicted with arrows, and data stores such as computer files are depicted with parallel lines. This notation conventional in this application area for showing the relations among processes and data flow. Within a bubble, however, the approach changes. Figure 3 (Best 1990, p.150) shows the internal structure of an update process. There is one of these for each of the master data files, and each is responsible for handling all possible updates to that data file.

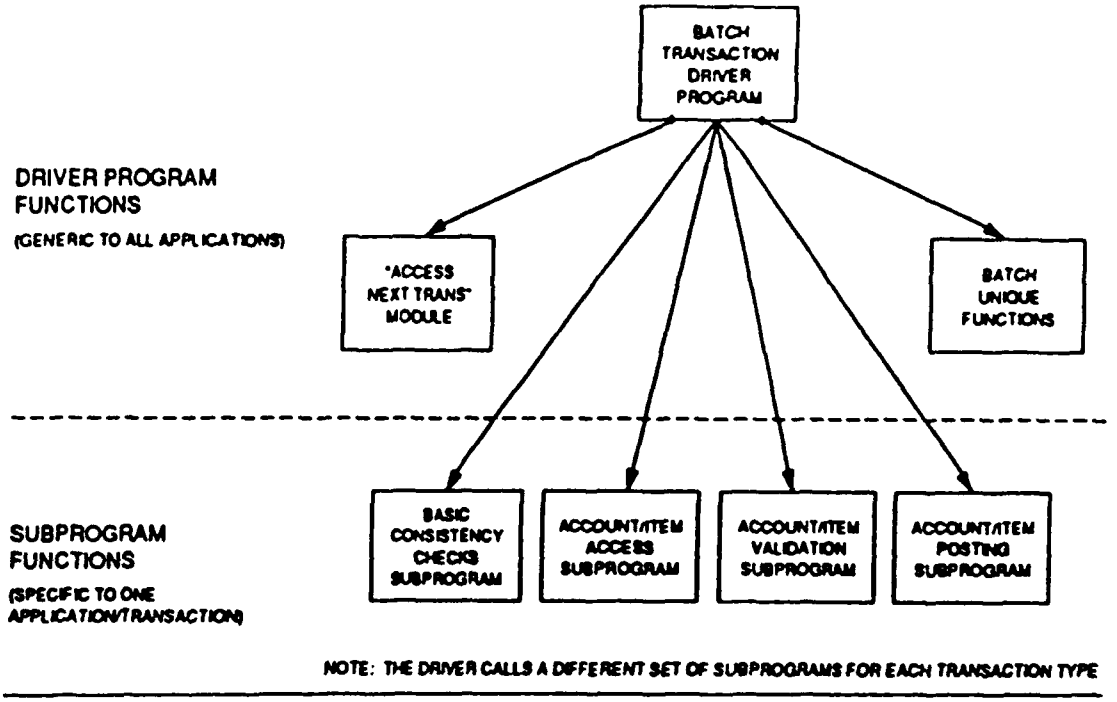

Figure 3: Internal structure of batch update process

In Figure 3, the boxes represent subprograms and the lines represent procedure calls. A single driver program processes all batch transactions. Each transaction has a standard set of subprograms that check the transaction request, access the required data, validate the transaction, and post the result. Thus all the program logic for each transaction is localized in a single set of subprograms. The figure indicates that the transaction-processing template is replicated so that each transaction has its own set. Note the difference even in graphical notation as the design focus shifts from the architecture to the code level.

The essential—batch sequential—parts of Figure 2 are redrawn in Figure 4 in a form that allows comparison to other architectures. The redrawn figure emphasizes the sequence of operations to be performed and the completion of each step before the start of its successor. It suppresses the on-line query support and updates to multiple master files, or databases.

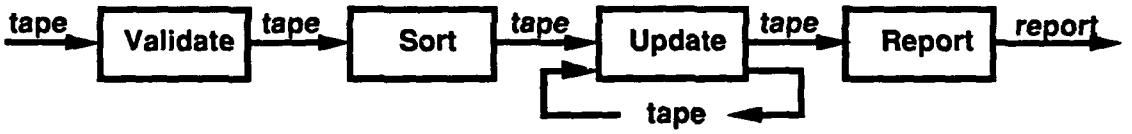

Figure 4: Batch sequential database architecture

2.2. Simple Repository

Two trends forced a change away from batch sequential processing. First, interactive technology provided the opportunity and demand for continuous processing of on-line updates as well as on-line queries. On-line queries of stale data are not very satisfactory; interaction requires incremental updates of the database, at least for on-line transactions (there is less urgency about transactions that arrive by slow means such as mail, since they have already incurred delays). Second, as organizations grew, the set of transactions and queries grew. Modifying a single large update program and a single large reporting program for each change to a transaction creates methodological bottlenecks. New types of processing were added often enough to discourage modification of a large update program for each new processing request. In addition, starting up large programs incurred substantial overheads at that time.

These trends led to a change in system organization. Figure 5 (Best 1990, p. 81) shows a "modern"—that is, interactive—system organization. The notation is as for Figure 2. This organization supports both interactive and batch processing for all transaction types; updates can occur continuously. Since

these are no longer periodic operations, the system also provides for periodic operations. Here, though, the transaction database and extract database are transient buffers; the account/item database is the central permanent store. The transaction database serves to synchronize multiple updates. The extract database solves a problem created by the addition of interactive processing—namely the loss of synchronization between the updating and reporting cycles. This figure obscures not only the difference between a significant database and a transient buffer but also the separation of transactions into separate processes.

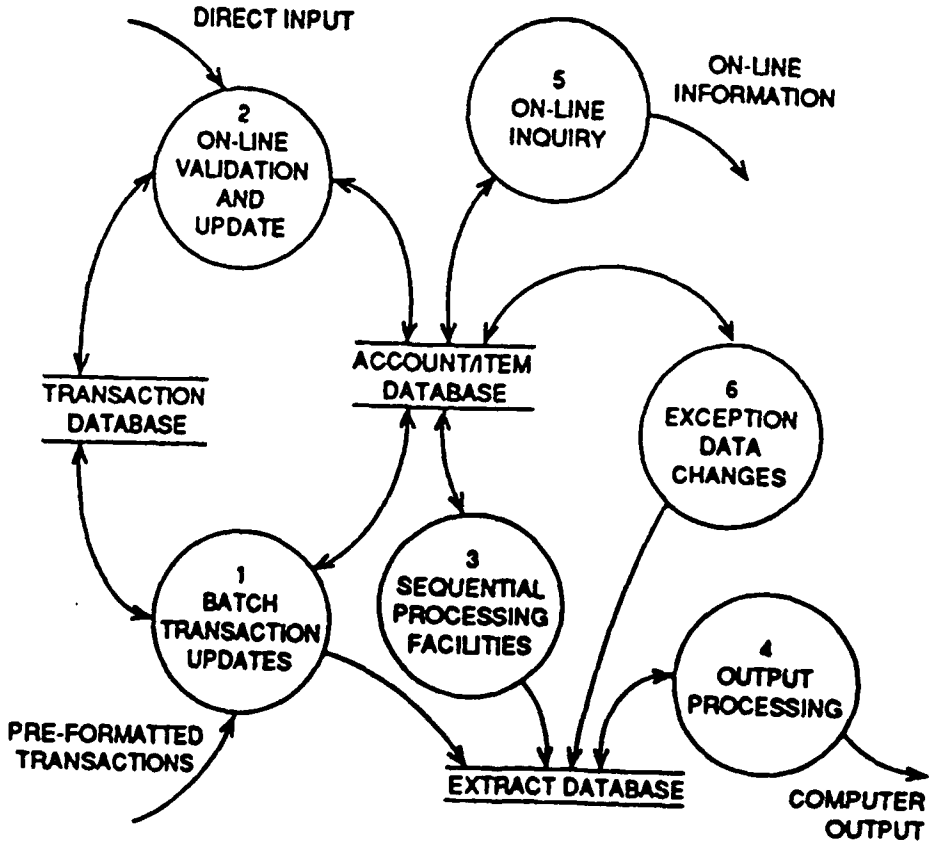

Figure 5: Dataflow diagram for interactive database

It is possible for transaction processing in this organization to resemble batch sequential processing. However, it is useful to separate the general overhead operations from the transaction-specific operations. It may also be useful to perform multiple operations on a single account all at once. Figure 6 (Best 1990, p.158) shows the program structure for the transactions in this new architecture. Since the transactions now exist individually rather than as alternatives within a single program, several of the bubbles in Figure 5 actually represent sets of independent bubbles.

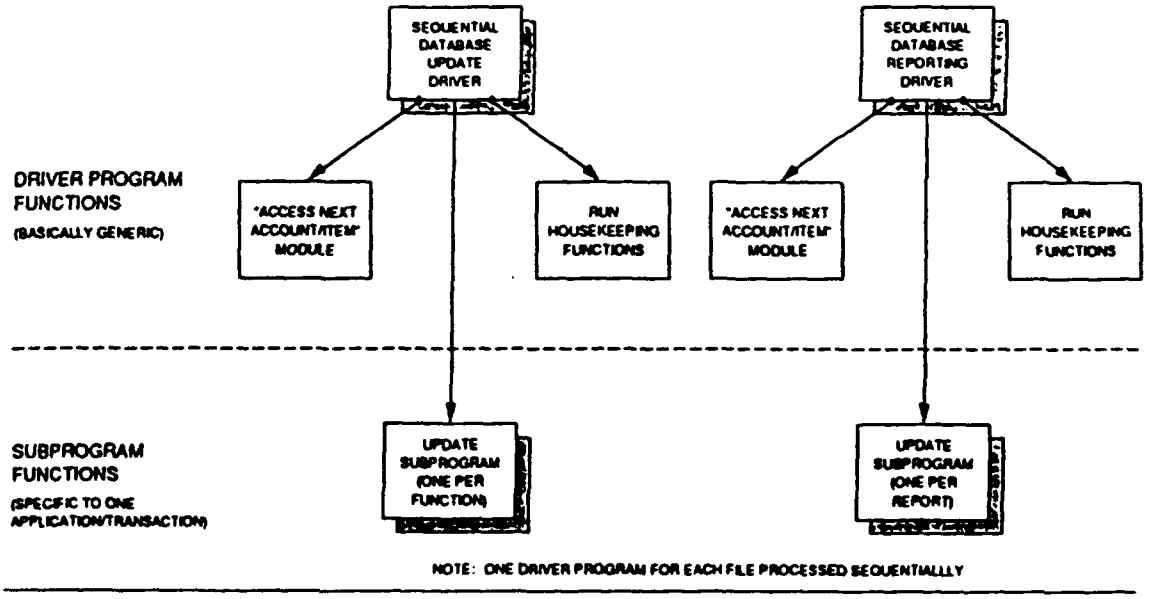

Figure 6: Internal structure of interactive update process

There is not a clean separation of architectural issues from coding issues in Figures 5 and 6. It is not unusual to find this, because explicit attention to the architecture as a separate level of software design is relatively recent. Indeed, Figures 5 and 6 suffer from information overload as well. The system structure is easier to understand if we first isolate the database updates. Figure 7 focuses narrowly on the database and its transactions. This is an instance of a fairly common architecture, a *repository*, in which shared persistent data is manipulated by independent functions each of which has essentially no permanent state. It is the core of a database system. Figure 8 adds two additional structures. The first is a control element that accepts the batch or interactive stream of transactions, synchronizes them, and selects which update or query operations to invoke, and in which order. This subsumes the transaction database of Figure 5. The second is a buffer that serves the periodic reporting function. This subsumes the extract database of Figure 5.

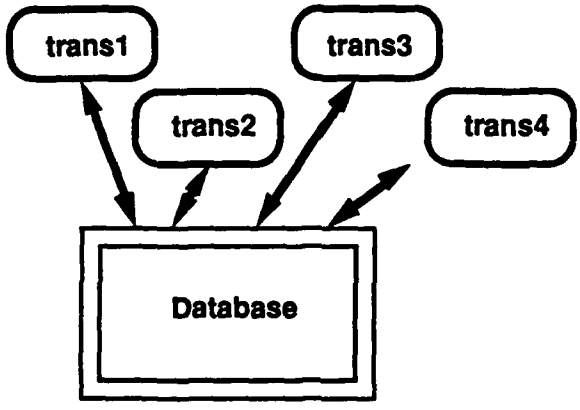

Figure 7: Simple repository database architecture

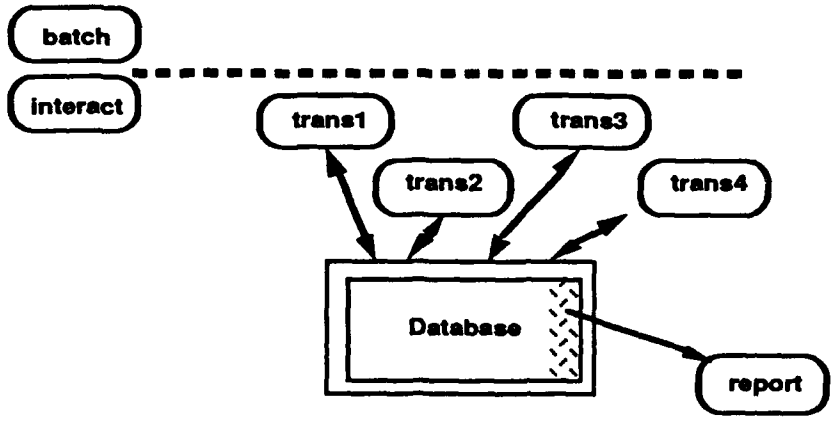

Figure 8: Repository architecture for database showing control and reporting

2.3. Virtual Repository

As organizations grew, databases came to serve multiple functions. Since this was usually a result of incremental growth, individual independent programs continued to be the locus of processing. In response, simple repositories gave way to databases that supported multiple views through schemas. Corporate reorganizations, mergers, and other consolidations of data forced the joint use of multiple databases. As a result, information could no longer be localized in a single database. Figure 9 (Kim and Seo 1991, p.13) gives a hint of the extent of the problem through the schemas that describe books in four libraries. Note, for example, that the call number is represented in different ways in all four schemas; in this case they're all Library of Congress numbers, so the more difficult case of a mixture of Library of Congress and Dewey numbering doesn't arise. Note also the assortment of ways the publisher's name, address, and (perhaps) telephone number are represented.

Library	Table name	Attributes	General description
CDB1: Main (Main Library)	item	(id*, title, author-name, subject, type, language)	Library items
	lc-num	(id*, c-letter, f-digit, s-digit, cuttering)	Library of Congress number
	publisher	(id*, name, tel, street, city, zip, state, country)	Publishers
	lend-info	(id*, lend-period, library-use-only, checked-out)	Lending information
	checkout-info	(id*, id-num, hour, day, month, year)	Borrower and due date
CDB2: Engineering (Engineering Library)	items	(id*, title, a-name, type, c-letter, f-digit, s-digit, cuttering)	Library items
	item-subject	(id*, subject)	Subject of each item
	item-language	(id#, language)	Language used in each item
	publisher	(id*, p-name, str-num, str-name, city, zip, state)	Publishers
	lend-info	(id*, lend-period, library-use-only, checked-out)	Lending information
	checkout-info	(id*, id-num, hour, day, month, year)	Borrower and due date
CDB3: City (City Public Library)	books	(id*, lc-num, name, title, subject)	Library items
	publisher	(id*, p-name, p-address)	Publishers
	lend-info	(id*, l-period, reference, checked-out)	Lending information
	checkout-info	(id*, dl-num, day, month, year)	Borrower and due date
CDB4: Comm (Community College Library)	item	(id*, lc-number, title, a-name)	Library items
	publisher-info	(id*, p#*, name, tel)	Publishers
	publisher-add	(p#*, st-num, st-name, room-num, city, state, zip)	Publisher address
	checkout-info	(id*, id, day, month, year)	Borrower and due date
	lc-num	(id*, category, user-name)	Library card number

*Indicates key attribute

Figure 9: Diversity of schemas for a single construct

Developing applications that rely on multiple diverse databases like these requires solution of two problems. First, the system must reconcile representation differences. Second, it must communicate results across distributed systems that may have not only different data representations but also different database schema representations. One approach to the unification of multiple schemas is called the federated approach. Figure 10 (Ahmed et al 1991, p. 21) shows one way to approach this, relying on the well-understood technology for handling multiple views on databases. The top of this figure shows how the usual database mechanisms integrate multiple schemas into a single schema. The bottom of the figure suggests an approach to importing data from autonomous external databases: For each database, devise a schema in its native schema language that exports the desired data and a matching schema in the schema language of the importer. This separates the solutions to the two essential problems and restricts the distributed system problem to communication between matching schemas.

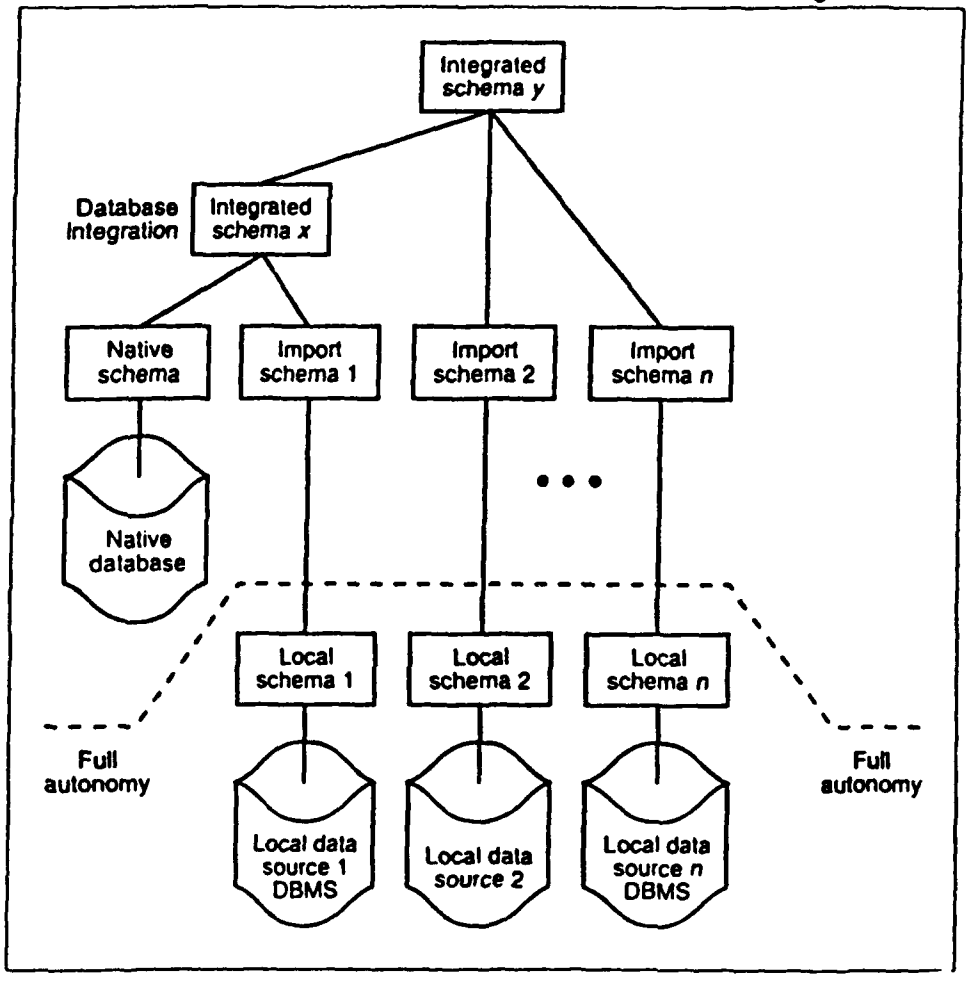

Figure 10: Combining multiple distributed schemas

Figure 10 combines solutions to two problems. Here again, the design is clearer if the discussion and diagram separate the two sets of concerns. Figure 11 shows the integration of multiple databases by unified schemas. It shows a simple composition of projections. The details about whether the data paths are local or distributed and whether the local schema and import schema are distinct are suppressed at this level of abstraction; these communication questions should be addressed in an expansion of the abstract filter design (and they may not need to be the same for all of the constituents).

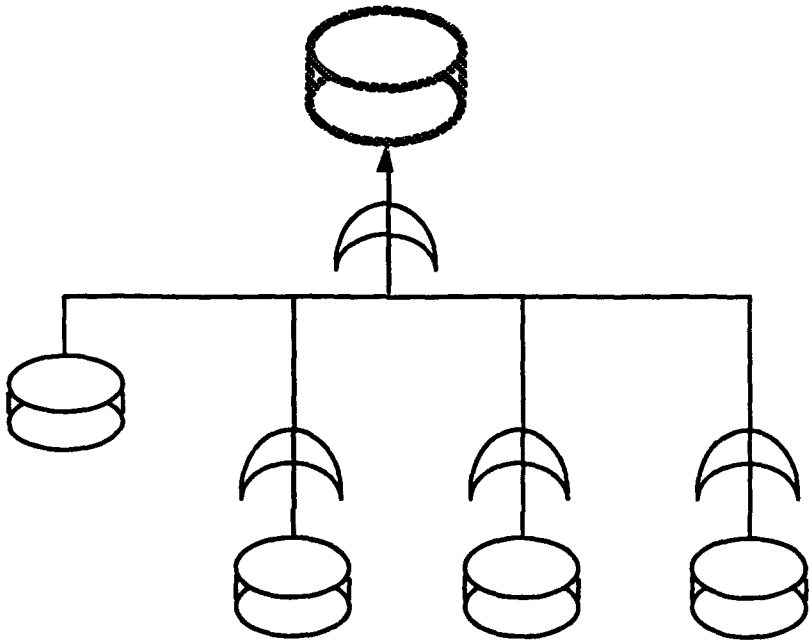

Figure 11: Integration of multiple databases

2.4. Hierarchical Layers

Unified schemas allow for merger of information, but their mappings are fixed, passive, and static. The designers of the views must anticipate all future needs; the mappings simply transform the underlying data; and there are essentially no provisions for recognizing and adapting to changes in the set of available databases. In the real world, each database serves multiple users, and indeed the set of users changes regularly. The set of available databases also changes, both because the population of databases itself changes and because network connectivity changes the set that is accessible. This exacerbates the usual problems of inconsistency across a set of databases. The commercial database community has begun to respond to this problem of dynamic reconfiguration. Distributed database products organized on a client-server model are beginning to challenge traditional mainframe database

management systems (Hovaness 1992). This set of problems is also of current interest in the database research community.

Figure 12 (Wiederhold 1992, p. 45) depicts one research scenario for active mediation between a constantly-changing set of users and a constantly-changing set of databases. Wiederhold proposes introducing active programs, called experts, to accept queries from users, recast them as queries to the available databases, and deliver appropriate responses to the users. These experts, or active mediators, localize knowledge about how to discover what databases are available and interact with them, about how to recast users' queries in useful forms, and about how to reconcile, integrate, and interpret information from multiple diverse databases.

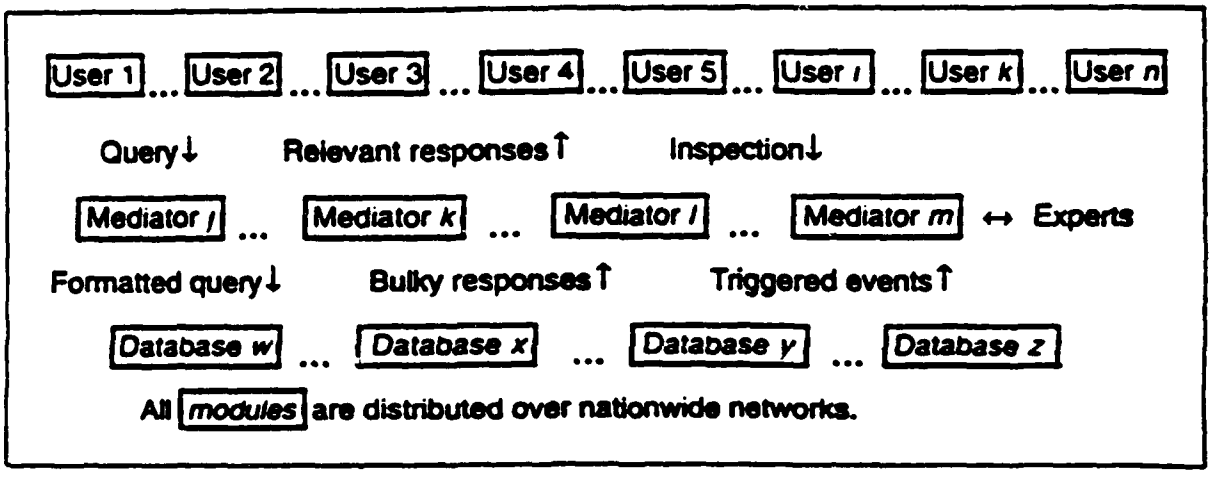

Figure 12: Multidatabase with mediators

In effect, Wiederhold's architecture uses *hierarchical layers* to separate the business of the users, the databases, and the mediators. The interaction between layers of the hierarchy will most likely be a *client-server* relation. This is not a repository because there is no enforced coherence of central shared data; it is not a batch sequential system (or any other form of pipeline) because the interaction with the data is incremental. Figure 13 recasts this in a form similar to the other examples.

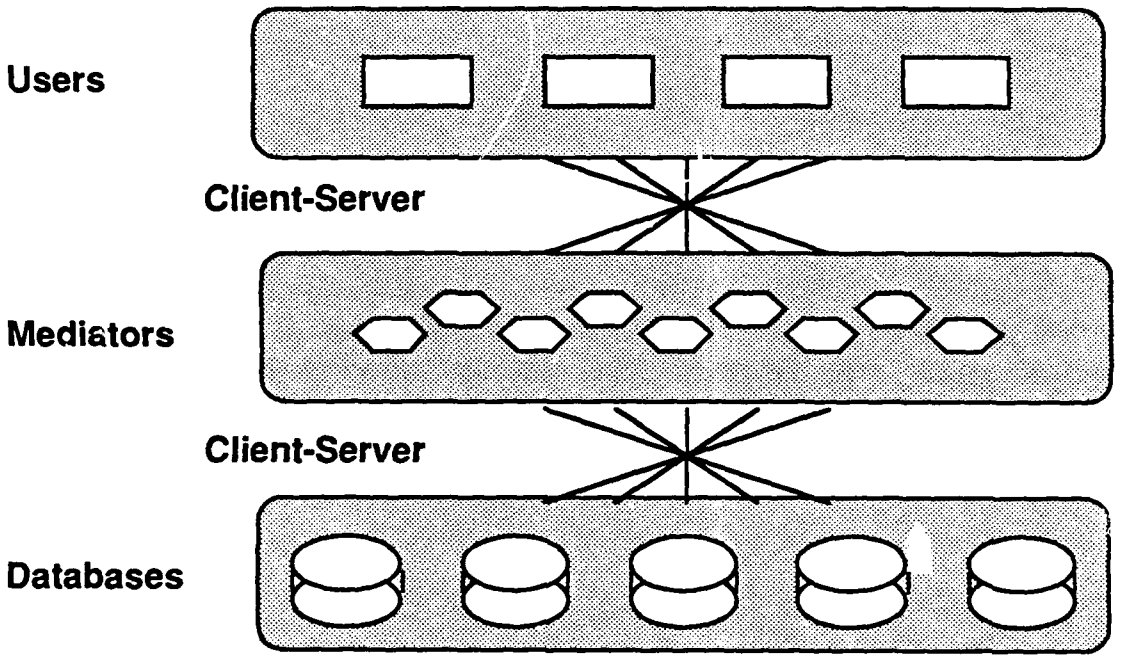

Figure 13: Layered architecture for multidatabase

2.5. Evolution of Shared Information Systems in Business Data Processing

These business data processing applications exhibit a pattern of development driven by changing technology and changing needs. The pattern was:

- *Batch processing:* Standalone programs; results are passed from one to another on magtape. *Batch sequential model.*
- *Interactive processing:* Concurrent operation and faster updates preclude batching, so updates are out of synchronization with reports. *Repository model* with external control.
- *Unified schemas:* Information becomes distributed among many different databases. One virtual repository defines (passive) consistent conversion mappings to multiple databases.

- *Multi-database:* Databases have many users; passive mappings don't suffice; active agents mediate interactions. *Layered hierarchy* with client-server interaction.

In this evolution, technological progress and expanding demand drive progress. Larger memories and faster processing enable access to an ever-wider assortment of data resources in a heterogeneous, distributed world. Our ability to exploit this remains limited by volume, complexity of mappings, the need to handle data discrepancies, and the need for sophisticated interpretation of requests for services and of available data.

3. Integration in Software Development Environments

Software development has relied on software tools for almost as long as data processing has relied on on-line databases. Initially these tools only supported the translation from source code to object code; they included compilers, linkers, and libraries. As time passed, many steps in the software development process became sufficiently routine to be partially or wholly automated, and tools now support analysis, configuration control, debugging, testing, and documentation as well. As with databases, the individual tools grew up independently. Although the integration problem has been recognized for nearly two decades (Toronto 1974), individual tools still work well together only in isolated cases.

3.1. Batch Sequential

The earliest software development tools were standalone programs. Often their output appeared only on paper and perhaps in the form of object code on cards or paper tape. Eventually most of the tools' results were at least in some magnetic—universally readable—form, but the output of each tool was most likely in the wrong format, the wrong units, or the wrong conceptual model for other tools to use. Even today, execution profiles are customarily provided in human-readable form but not propagated back to the compiler for optimization. Effective sharing of information was thus limited by lack of knowledge about how information was encoded in representations. As a result, manual translation of one tool's output to another tool's input format was common.

As time passed, new tools incorporated prior knowledge of related tools, and the usefulness of shared information became more evident. Scripts grew up to invoke tools in fixed orders. These scripts essentially defined batch sequential architectures.

This remains the most common style of integration for most environments. For example, in unix both shell scripts and make follow this paradigm. ASCII text is the universal exchange representation, but the conventions for encoding internal structure in ASCII remain idiosyncratic.

3.2. Transition from Batch Sequential to Repository

Our view of the architecture of a system can change in response to improvements in technology. The way we think about compilers illustrates this. In the 1970s, compilation was regarded as a sequential process, and the organization of a compiler was typically drawn as in Figure 14. Text enters at

the left end and is transformed in a variety of ways—to lexical token stream, parse tree, intermediate code—before emerging as machine code on the right. We often refer to this compilation model as a pipeline, even though it was (at least originally) closer to a batch sequential architecture in which each transformation ("pass") ran to completion before the next one started.

Figure 14: Traditional compiler model

In fact, even the batch sequential version of this model was not completely accurate. Most compilers created a separate symbol table during lexical analysis and used or updated it during subsequent passes. It was not part of the data that flowed from one pass to another but rather existed outside all the passes. So the system structure was more properly drawn as in Figure 15.

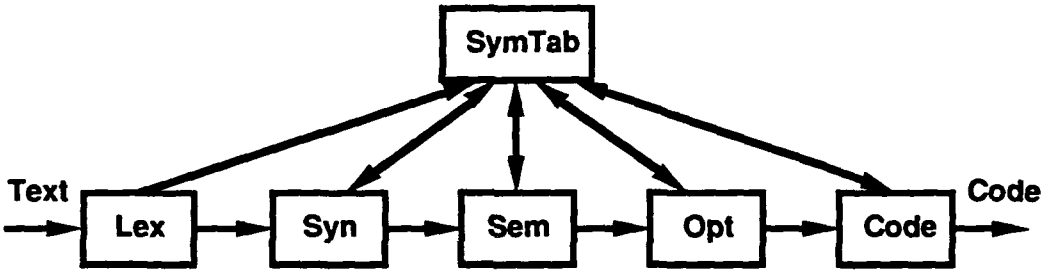

Figure 15: Traditional compiler model with symbol table

As time passed, compiler technology grew more sophisticated. The algorithms and representations of compilation grew more complex, and increasing attention turned to the intermediate representation of the program during compilation. Improved theoretical understanding, such as attribute grammars, accelerated this trend. The consequence was that by the mid-1980s the intermediate representation (for example, an attributed parse tree), was the center of attention. It was created early during compilation, manipulated during the remainder, and discarded at the end. The data structure might change in detail, but it remained substantially one growing structure throughout. However, we continued (sometimes to the present) to model the compiler with sequential data flow as in Figure 16.

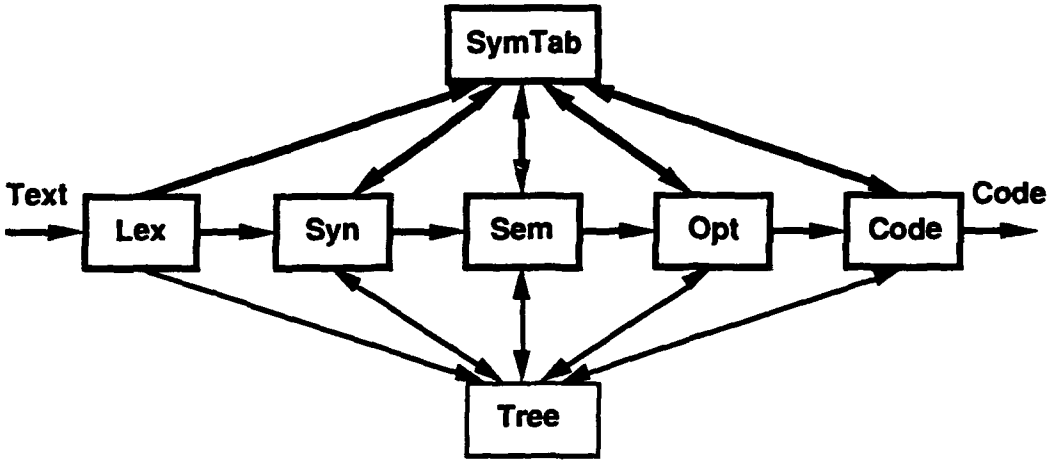

Figure 16: Modern canonical compiler

In fact, a more appropriate view of this structure would re-direct attention from the sequence of passes to the central shared representation. When you declare that the tree is the locus of compilation information and the passes define operations on the tree, it becomes natural to re-draw the architecture as in Figure 17. Now the connections between passes denote control flow, which is a more accurate depiction; the rather stronger connections between the passes and the tree/symbol table structure denote data access and manipulation. In this fashion, the architecture has become a repository, and this is indeed a more appropriate way to think about a compiler of this class.

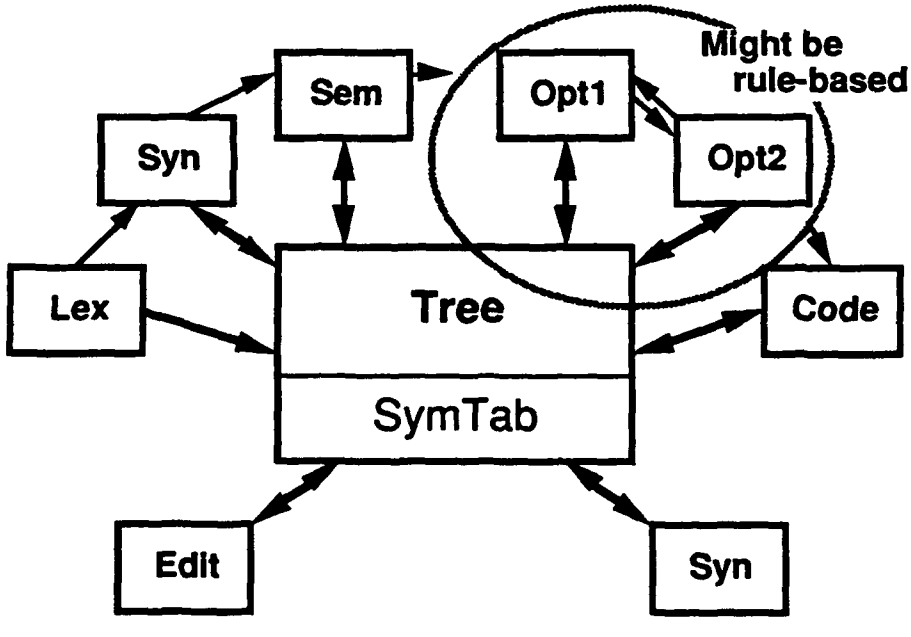

Figure 17: Repository view of modern compiler

Happily, this new view also accommodates various tools that operate on the internal representation rather than the textual form of a program; these include syntax-directed editors and various analysis tools.

Note that this repository resembles the database repository in some respects and differs in others. Like the database, the information of the compilation is localized in a central data component and operated on by a number of independent computations that interact only through the shared data. However, whereas the execution order of the operations in the database was determined by the types of the incoming transactions, the execution order of the compiler is predetermined, except possibly for opportunistic optimization.

3.3. Repository

Batch sequential tools and compilers—even when organized as repositories—do not retain information from one use to another. As a result, a body of knowledge about the program is not accumulated. The need for auxiliary information about a program to supplement the various source, intermediate, and object versions became apparent, and tools started retaining information about the prior history of a program.

The repository of the compiler provided a focus for this data collection. Efficiency considerations led to incremental compilers that updated the previous version of the augmented parse tree, and some tools came to use this shared representation as well. Figure 18 shows some of the ways that tools could interact with a shared repository.

- *Tight coupling:* Share detailed knowledge of the common, but proprietary, representation among the tools of a single vendor
- *Open representation:* Publish the representation so that tools can be developed by many sources. Often these tools can manipulate the data, but they are in a poor position to change the representation for their own needs.
- *Conversion boxes*: Provide filters that import or export the data in foreign representations. The tools usually lose the benefits of incremental use of the repository.
- *No contact:* Prevent a tool from using the repository, either explicitly, through excess complexity, or through frequent changes.

These alternatives have different functional, efficiency, and market implications.

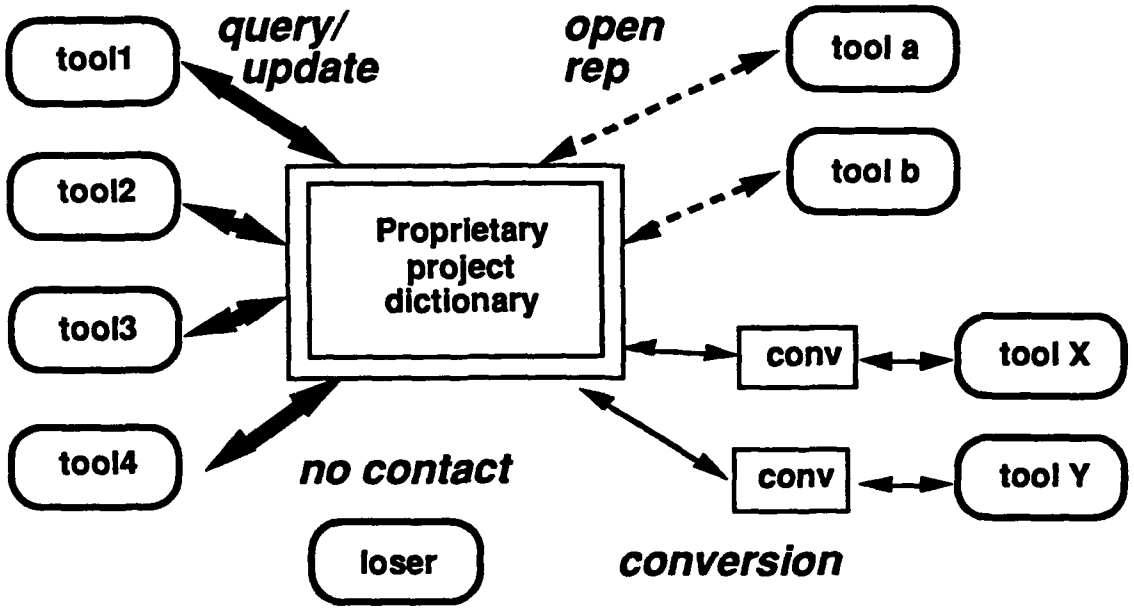

Figure 18: Software tools with shared representation

3.4. Hierarchical Layers

Current work on integration emphasizes interoperability of tools, especially in distributed systems. Figure 19 (Chen and Norman, 1992, p.19) shows one approach, the NIST/ECMA reference model. It resembles in some ways the layered architecture with mediators for databases, but it is more elaborate because it attempts to integrate communications and user interfaces as well as representation. It also embeds knowledge of software development processes, such as the order in which tools must be used and what situations call for certain responses.

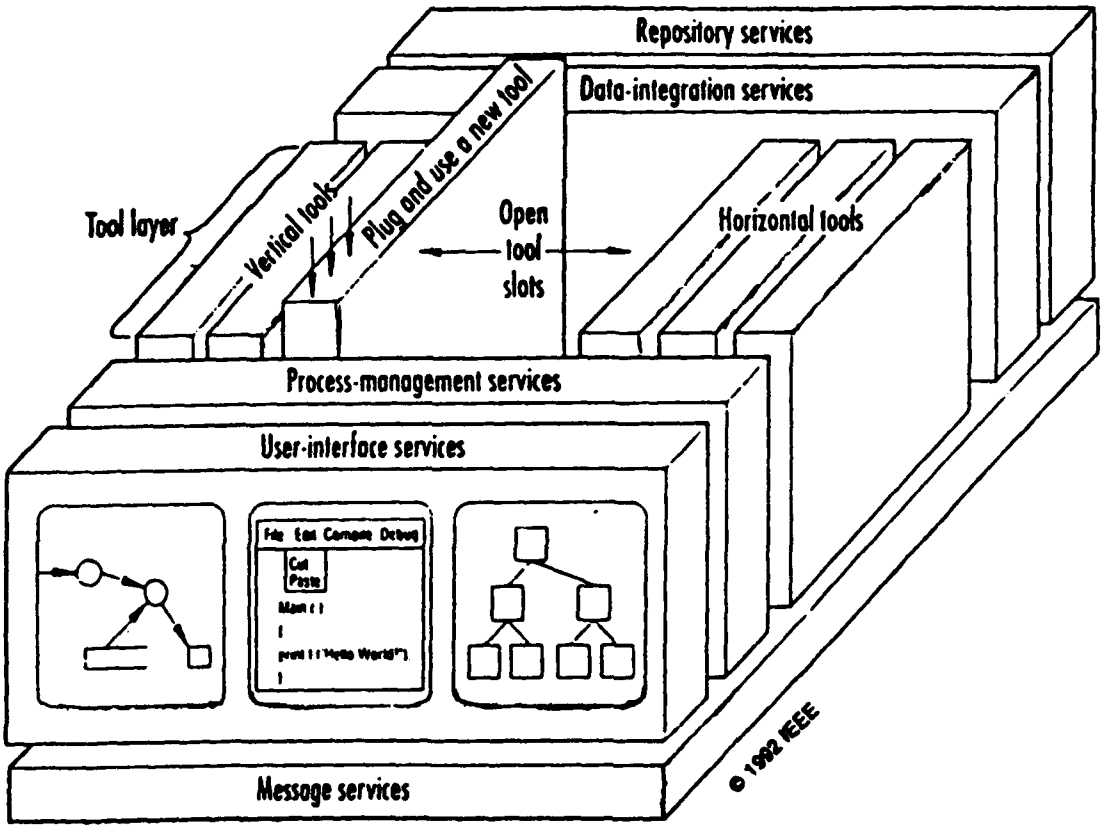

Figure 19: NIST/ECMA reference model for environment integration

Note, however, that whereas this model provides for integration of data, it provides communication and user interface services directly. That is, this model allows for integration of multiple representations but fixes the models for user interfaces and communication.

In one variation on the integrated-environment theme, the integration system defined a set of "events" (e.g., "module foo.c recompiled") and provides support for tools to announce or to receive notice of the occurrence of events. This provides a means of communicating the need for action, but it does not solve the central problem of sharing information.

3.5. Evolution of Shared Information Systems in Software Development Environments

Software development ha ifferent requirements from database processing. As compared to databases, oftware development involves more different types of data, fewer instances of each distinct type, and slower query rates. The units of information are larger, more complex, and less discrete than in traditional databases. The lifetime of software development information, however, is not (or at least should not be) shorter than database lifetimes.

Despite the differences in application area and characteristics of the supporting data, the essential problem of collecting, storing, and retrieving shared data about an ongoing process is common to the two areas. It is therefore not surprising to find comparable evolutionary stages in their architectures.

Here the forces for evolution were

- the advent of on-line computing, which drove the shift from batch to interactive processing for many functions
- the concern for efficiency, which is driving a reduction in the granularity of operations, shifting from complete processing of systems to processing of modules to incremental development
- the need for management control over the entire software development process, which is driving coverage to increase from compilation to the full life cycle

Integration in this area is still incomplete. Data conversions are passive, and the ordering of operations remains relatively rigid. The integration systems can exploit only relatively coarse system information, such as file and date. Software development environments are under pressure to add capabilities for handling complex dependencies and selecting which tools to use. Steps toward more sophistication show up in the incorporation of metamodels to

describe sharing, distribution, data merging, and security policies. The process-management services of the NIST/ECMA model are not yet well developed, and they will initially concentrate on project-level support. But integration across all kinds of information and throughout the life cycle is on the agenda, and intelligent assistance is often mentioned on the wish-list.

4. Integration in Building Design

The previous two examples come from the information technology fields. For the third example we turn to an application area, the building construction industry. This industry requires a diverse variety of expertise. Distinct responsibilities correspond to matching sets of specialized functions. Indeed, distinct subindustries support these specialties. A project generally involves a number of independent, geographically dispersed companies. The diversity of expertise and dispersion of the industry inhibit communication and limit the scope of responsibilities. Each new project creates a new coalition, so there is little accumulated shared experience and no special advantage for pairwise compatibility between companies. However, the subtasks interact in complex, sometimes non-obvious ways, and coordination among specialties (global process expertise) is itself a specialty (Terk 1992).

The construction community operates on divide-and-conquer problem solving with interactions among the subproblems. This is naturally a distributed approach; teams independent subcontractors map naturally to distributed problem-solving systems with coarse-grained cooperation among specialized agents. However, the separation into subproblems is forced by the need for specialization and the nature of the industry; the problems are not inherently decomposable, and the subproblems are often interdependent.

In this setting it was natural for computing to evolve bottom-up. Building designers have exploited computing for many years for tasks ranging from accounting to computer-aided design. We are concerned here with the software that performs analysis for various stages of the design activity. The 1960s and 1970s saw a number of algorithmic systems directed at aiding in the performance of individual phases of the facility development. However, a large number of tasks in facility development depend on judgment, experience, and rules of thumb accumulated by experts in the domain. Such tasks cannot be performed efficiently in an algorithmic manner (Terk 1992).

The early stages of development, involving standalone programs and batch-sequential compositions, are sufficiently similar to the two previous examples that it is not illuminating to review them. The first steps toward integration focused on support-supervisory systems, which provided basic services such as data management and information flow control to individual independent applications, much as software development environments did. The story picks up from the point of these early integration efforts.

Integrated environments for building design are frameworks for controlling a collection of standalone applications that solve part of the building design problem (Terk 1992). They must be

- efficient in managing problem-solving and information exchange
- flexible in dealing with changes to tools
- graceful in reacting to changes in information and problem solving strategies

These requirements derive from the lack of standardized problem-solving procedures; they reflect the separation into specialties and the geographical distribution of the facility development process.

4.1. Repository

Selection of tools and composition of individual results requires judgment, experience, and rules of thumb. Because of coupling between subproblems it is not algorithmic, so integrated systems require a planning function. The goal of an integrated environment is integration of data, design decisions, and knowledge. Two approaches emerged: the closely-coupled Master Builder, or monolithic system, and the design environment with cooperating tools. These early efforts at integration added elementary data management and information flow control to a tool-set.

The common responsibilities of a system for distributed problem-solving are:

- Problem partitioning (divide into tasks for individual agents)
- Task distribution (assign tasks to agents for best performance)
- Agent control (strategy that assures tasks are performed in organized fashion)
- Agent communication (exchange of information essential when subtasks interact or conflict)

The construction community operates on divide-and-conquer problem solving with interactions among the subproblems. This is naturally a distributed approach; teams independent subcontractors map naturally to distributed problem-solving systems with coarse-grained cooperation among specialized agents. However, the nature of the industry—its need for specialization—forces the separation into subproblems; the problems are not inherently decomposable, and the subproblems are often interdependent. This raises the control component to a position of special significance.

Terk (1992) surveyed and classified many of the integrated building design environments that were developed in the 1980s. Here's what he found:

- *Data:* mostly repositories: shared common representation with conversions to private representations of the tools
- *Communication:* mostly shared data, some messaging
- *Tools:* split between closed (tools specifically built for this system) and open (external tools can be integrated)
- *Control:* mostly single-level hierarchy; tools at bottom; coordination at top
- *Planning:* mostly fixed partitioning of kind and processing order; scripts sometimes permit limited flexibility

So the typical system was a repository with a sophisticated control and planning component. A fairly typical such system, IBDE (Fenves et al 1990) appears in Figure 20. Although the depiction is not typical, the distinguished position of the global data shows clearly the repository character. The tools that populate this IBDE are

- ARCHPLAN develops architectural plan from site, budget, geometric constraints
- CORE lays out building service core (elevators, stairs, etc.)
- STRYPES configures the structural system (e.g., suspension, rigid frame, etc.)
- STANLAY performs preliminary structural design and approximate analysis of the structural system.
- SPEX performs preliminary design of structural components.
- FOOTER designs the foundation.
- CONSTRUCTION PLANEX generates construction schedule and estimates cost.

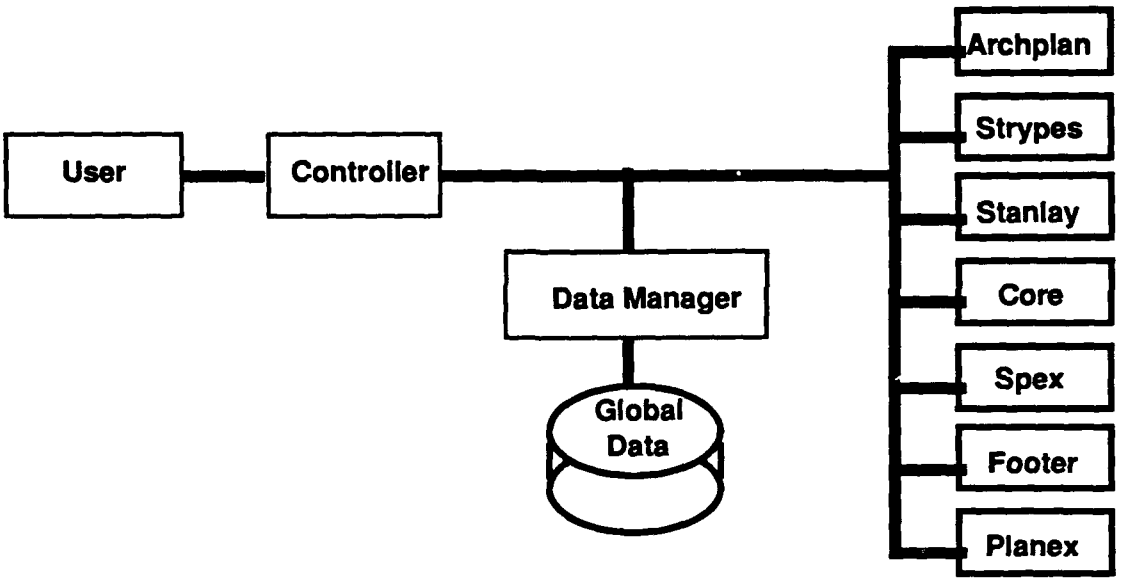

Figure 20: Integrated building design environment

4.2. Intelligent Control

As integration and automation proceed, the complexity of planning and control grows to be a significant problem. Indeed, as this component grows more complex, its structure starts to dominate the repository structure of the data. The difficulty of reducing the planning to pure algorithmic form makes this application a candidate for intelligent control.

The Engineering Design Research Center at CMU is exploring the development of intelligent agents that can learn to control external software systems, or systems intended for use with interactive human intervention. Integrated building design is one of the areas they have explored. Figure 22 (Newell and Steier 1991) shows their design for an intelligent extension of the original IBDE system, Soar/IBDE. That figure is easier to understand in two stages, so Figure 21 shows the relation of the intelligent agent to the external software systems before Figure 22 adds the internal structure of the intelligent agent. Figure 21 is clearly derived from Figure 20, with the global data moved to the status of just another external software system. The emphasis in Soar/IBDE was control of the interaction with the individual agents of IBDE.

From the standpoint of the designer's general position on intelligent control this organization seems reasonable, as the agent is portrayed as interacting with whatever software is provided. However, the global data plays a special role in this system. Each of the seven other components must interact with the global data (or else it makes no sense to retain the global data). Also, the intelligent

agent may also find that the character of interaction with the global data is special, since it was designed to serve as a repository, not to interact with humans. Future enhancements of this system will probably need to address the interactions among components as well as the components themselves.

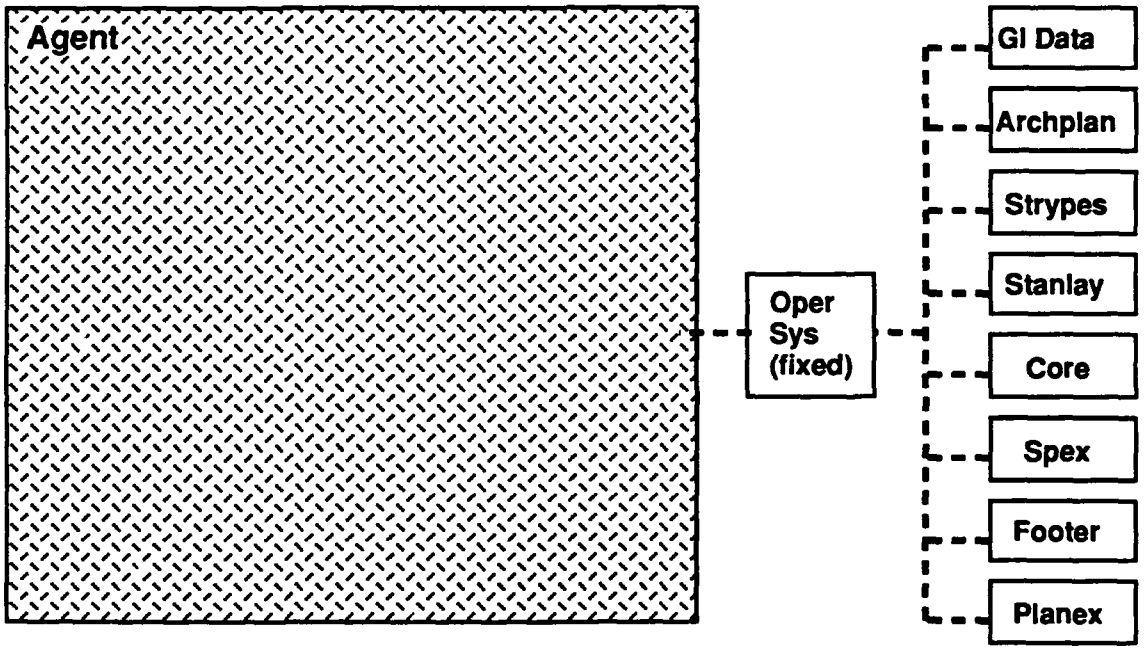

Figure 21: High-level architecture for intelligent IBDE

Figure 22 adds the fine structure of the intelligent agent. The agent has six major components. It must be able to identify and formulate subtasks for the set of external software systems and express them in the input formats of those systems. It must receive the output and interpret it in terms of a global overview of the problem. It must be able to understand the actions of the components as they work toward solution of the problem, both in terms of general knowledge of the task and specific knowledge of the capabilities of the set of external software systems.

The most significant aspect of this design is that the seven external software systems are interactive. This means that their input and output are incremental, so a component that needs to understand their operation must retain and update a history of the interaction. The task becomes vastly more complex when pointer input and graphical output are included, though this is not the case in this case.

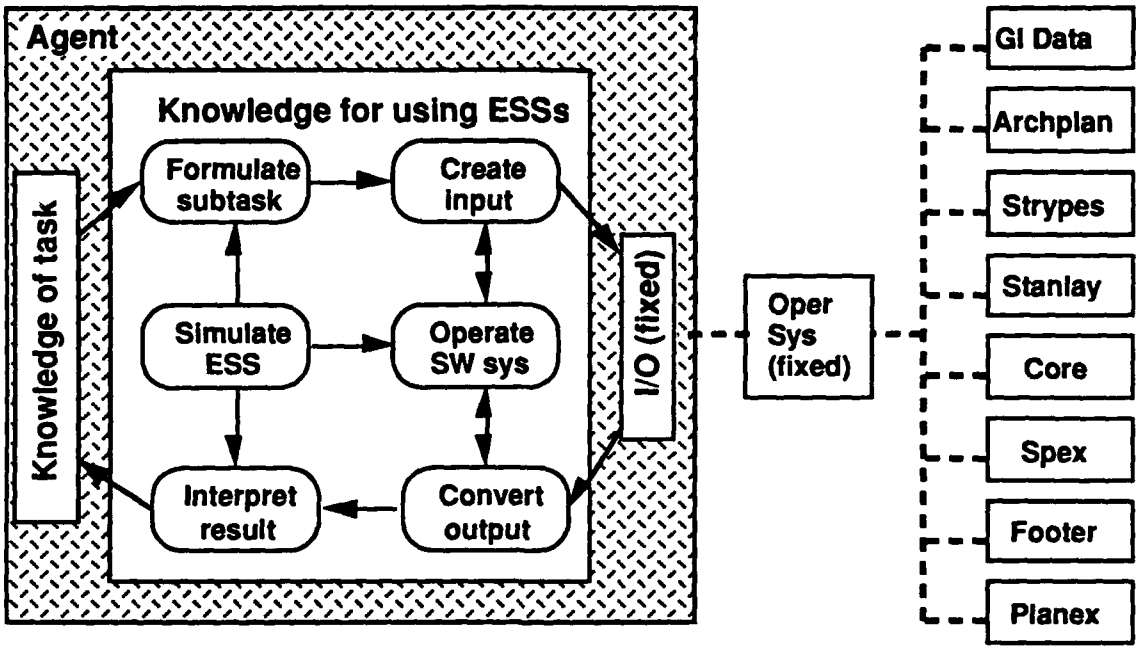

Figure 22: Detailed architecture for Soar/IBDE

4.3. Evolution of Shared Information Systems in Building Design

Integration in this area is less mature than in databases and software development environments. Nevertheless, the early stages of integrated building or facility environments resemble the early stages of the first two examples. The evolutionary shift to layered hierarchies seems to come when many users must select from a diverse set of tools and they need extra system structure to coordinate the effort of selecting and managing a useful subset. These systems have not reached this stage of development yet, so we don't yet have information on how that will emerge.

In this case, however, the complexity of the task makes it a prime candidate for intelligent control. This opens the question of whether intelligent control could be of assistance in the other two examples, and if so what form it will take. The single-agent model developed for Soar/IBDE is one possibility, but the enrichment of database mediators to make them able of independent intelligent action (like knowbots) is clearly another.

5. Architectural Structures for Shared Information Systems

While examining examples of software integration, we have seen a variety of general architectural patterns, or idioms for software systems. In this section we re-examine the data flow and repository idioms to see the variety that can occur within a single idiom.

Current software tools do not distinguish among different kinds of components at this level. These tools treat all modules equally, and they mostly assume that modules interact only via procedure calls and perhaps shared variables. By providing only a single model of component, they tend to blind designers to useful distinctions among modules. Moreover, by supporting only a fixed pair of low-level mechanisms for module interaction, they tend to blind designers to the rich classes of high-level interactions among components. These tools certainly provide little support for documenting design intentions in such a way that they become visible in the resulting software artifacts.

By making the richness of these structures explicit, we focus the attention of designers on the need for coherence and consistency of the system's design. Incorporating this information explicitly in a system design should provide a record that simplifies subsequent changes and increases the likelihood that later modifications will not compromise the integrity of the design. The architectural descriptions focus on design issues such as the gross structure of the system, the kinds of parts from which it is composed, and the kinds of interactions that take place.

The use of well-known patterns leads to a kind of reuse of design templates. These templates capture intuitions that are a common part of our folklore: it is now common practice to draw box-and-line diagrams that depict the architecture of a system, but no uniform meaning is yet associated with these diagrams. Many anecdotes suggest that simply providing some vocabulary to describe parts and patterns is a good first step.

By way of recapitulation, we now examine variations on two of the architectural forms that appear above: data flow and repositories.

5.1 Variants on Data Flow Systems

The data flow architecture that repeatedly occurs in the evolution of shared information systems is the batch sequential pattern. However, the most familiar example of this genre is probably the unix pipe-and-filter system. The similarity

of these architectures is apparent in the diagrams used for systems of the respective classes, as indicated in Figure 23. Both decompose a task into a (fixed) sequence of computations. They interact only through the data passed from one to another and share no other information. They assume that the components read and write the data as a whole—that is, the input or output contains one complete instance of the result in some standard order. There are differences, though. Batch sequential systems are

- very coarse-grained
- unable to do feedback in anything resembling real time
- unable to exploit concurrency
- unlikely to proceed at an interactive pace

On the other hand, pipe-and-filter systems are

- fine-grained, beginning to compute as soon as they consume a few input tokens
- able to start producing output right away (processing is localized in the input stream)
- able to perform feedback (though most shells can't express it)
- often interactive

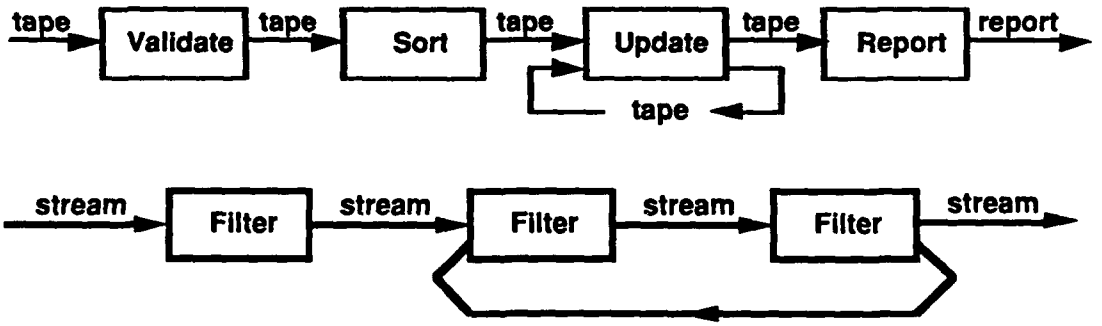

Figure 23 a, b: Comparison of (a) batch sequential and (b) pipe/filter architectures

5.2. Variants on Repositories

The other architectural pattern that figured prominently in our examples was the repository. Repositories in general are characterized by a central shared data store coupled tightly to a number of independent computations, each with its own expertise. The independent computations interact only through the shared data, and they do not retain any significant amount of private state. The variations differ chiefly in the control apparatus that controls the order in which

the computations are invoked, in the access mechanisms that allow the computations access to the data, and in the granularity of the operations.

Figures 7 and 8 show a database system. Here the control is driven by the types of transactions in the input stream, the access mechanism is usually supported by a specialized programming language, and the granularity is that of a database transaction.

Figure 17 shows a programming language compiler. Here control is fixed (compilation proceeds in the same order each time), the access mechanism may be full conversion of the shared data structure into an in-memory representation or direct access (when components are compiled into the same address space), and the granularity is that of a single pass of a compiler.

Figure 18 shows a repository that supports independent tools. Control may be determined by direct request of users, or it may in some cases be handled by an event mechanism also shared by the tools. A variety of access methods are available, and the granularity is that of the tool set.

One prominent repository has not appeared here; it is mentioned now for completeness—to extend the comparison of repositories. This is the blackboard architecture, most frequently used for signal-processing applications in artificial intelligence (Nii 1986) and depicted in Figure 24. Here the independent computations are various knowledge sources that can contribute to solving the problem—for example, syntactic-semantic connection, phoneme recognition, word candidate generation, and signal segmentation for speech understanding. The blackboard is a highly-structured representation especially designed for the representations pertinent to the application. Control is completely opportunistic, driven by the current state of the data on the blackboard. The abstract model for access is direct visibility, as of many human experts watching each other solve a problem at a real blackboard (understandably, implementations support this abstraction with more feasible mechanisms). The granularity is quite fine, at the level of interpreting a signal segment as a phoneme.

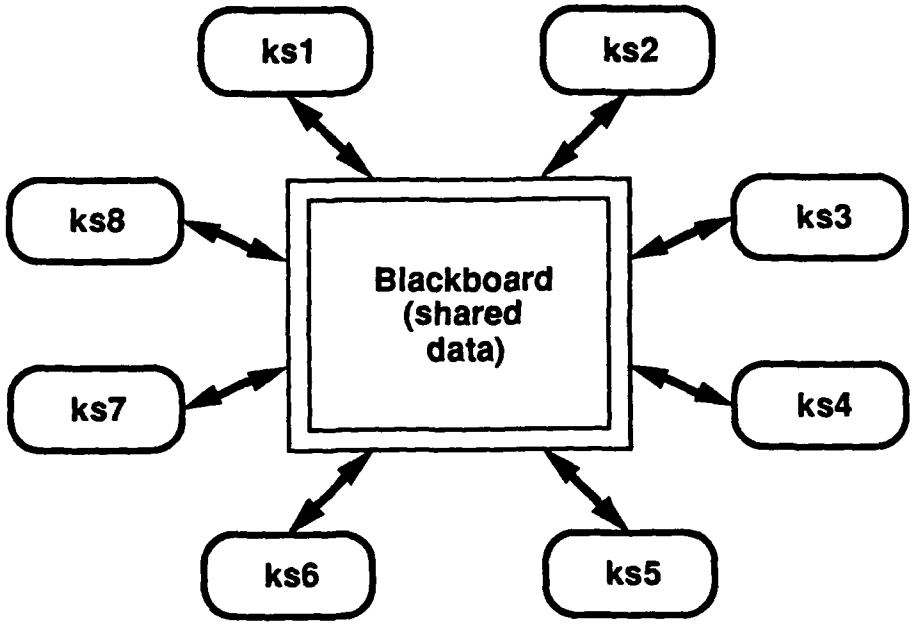

Figure 24: Blackboard architecture

6. Conclusions

Three tasks arising in different communities deal with collecting, manipulating, and preserving shared information. In each case changing technologies and requirements drove changes in the architectural form commonly used for the systems. We can identify that sequence as a common evolutionary pattern for shared information systems:

- isolated applications without interaction
- batch sequential processing
- repositories for integration via shared data
- layered hierarchies for dynamic integration across distributed systems

Since problems remain and new technology continues to emerge, this pattern may grow in the future, for example to add active control by intelligent agents.

These examples show one case in which a common problem structure appears in several quite different application areas. This suggests that attempts to exploit "domain knowledge" in software design should characterize domains by their computational requirements—e.g., shared information systems—as well as by industry—e.g., data processing, software development, or facility design. In addition, the examples show that within a single domain, differences among requirements or operational settings may change the preferred architecture. Taken together, this suggests that the notion of a single domain-specific architecture serving a segment of an industry may not fully exploit our growing architectural capabilities.

The models, notations, and tools for specifying software architectures remain informal. Although even informal models are useful, research in several areas is required to make these more precise and robust.

- Complete a taxonomy of common architectural patterns.
- Define and implement better abstractions for the interactions among components; at present system descriptions are cast in terms of procedure calls no matter what the abstractions may be.
- Establish ways to encapsulate stand-alone systems and express the resulting interfaces so they can be used as subsystems; linguistically this is a closure problem.
- Continue the exploration of independent agents for integration, especially in dynamically changing distributed systems.

Acknowledgments

This paper was written in honor of Allen Newell. Although we never collaborated directly, our professional interests touched from time to time, and these encounters often shaped the subsequent path of my research. This paper was so shaped at both ends: we discussed design levels many years ago, and I became acquainted with Soar/IBDE only a year or so ago. I also owe thanks to David Steier for discussions about Soar/IBDE and to David Garlan for ongoing collaboration in software architectures.

Several figures are reprinted by permission: Figure 1 from (Bell and Newell 1971), Gordon Bell. Figures 2, 3, 5, and 6 from (Best 1990) copyright © 1990 John Wiley and Sons Inc. Figures 9 from (Kim and Seo 1991), 10 from (Ahmed et al 1991), 12 from (Wiederhold 1992), and 19 from (Chen and Norman 1992) copyright © 1991, 1992 IEEE. Figures 20 and Figure 22 from (Newell and Steier), David Steier.

References

(Ahmed et al 1991) Rafi Ahmed et al. "The Pegasus Heterogeneous Multidatabase System." *IEEE Computer*, December 1991, 24, 12, pp. 19-27.

(Bell and Newell 1971) C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples.* McGraw-Hill 1971.

(Best 1990) Laurence J. Best. *Application Architecture: Modern Large-Scale Information Processing.* Wiley 1990.

(Chen and Norman 1992) Minder Chen and Ronald J. Norman. "A Framework for Integrated CASE." *IEEE Software*, March 1992, 9, 2, pp. 18-22.

(CSTB 1992) Computer Science and Telecommunications Board. *Keeping the US Computer Industry Competitive: Systems Integration.* National Academy Press 1992.

(Fenves et al 1990) S. J. Fenves et al. "An Integrated Software Environment for Building Design and Construction." *Computer-Aided Design*, 22, 1, pp. 27-36.

(Garlan and Shaw 1993) David Garlan and Mary Shaw. "An Introduction to Software Architecture." In V. Ambriola and G. Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering*, I, World Scientific Publishing Company, 1993 (to appear).

(Hovaness 1992) Haig Hovaness. "Price War: There's Fierce Combat Ahead over the Cost of Client-Server Databases." *Corporate Computing*, December 1992, 1, 6, pp. 45-46.

(Kim and Seo 1991) Won Kim and Jungyun Seo. "Classifying Schematic and Data Heterogeneity in Multidatabase Systems." *IEEE Computer*, December 1991, 24, 12, pp. 12-18.

(Newell 1982) Allen Newell. "The Knowledge Level." *Artificial Intelligence*, 1982, 18, pp. 87-127.

(Newell 1990) Allen Newell. *Unified Theories of Cognition.* Harvard University Press 1990.

(Newell and Steier 1991) Allen Newell and David Steier. "Intelligent Control of External Software Systems." *AI in Engineering*, to appear (was Carnegie Mellon University, Engineering Design Research Center Report EDRC 05-55-91).

(Nii 1986) H. Penny Nii. "Blackboard Systems." *AI Magazine*, 1986, 7, 3, pp. 38-53 and 7, 4, pp. 82-107.

(Nilsson et al 1990) Erik G. Nilsson et al. "Aspects of Systems Integration." *Systems Integration '90, Proc. First International Conference on Systems Integration*, 1990, pp. 434-443.

(Terk 1992) Michael Terk. A Problem-Centered Approach to Creating Design Environments for Facility Development. PhD Thesis, Civil Engineering Department, Carnegie Mellon University, 1992.

(Toronto 1974) *Proceedings of Workshop on the Attainment of Reliable Software.* University of Toronto, Toronto, Canada, June 1974.

(Wiederhold 1992) Gio Wiederhold. "Mediators in the Architecture of Future Information Systems." *IEEE Computer*, March 1992, 25, 3, pp. 38-48.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-93-TR-3	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-93-180
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) HQ ESC/ENS Hanscom Air Force Base, MA 01731-2116
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63756E; PROJECT NO.: N/A; TASK NO: N/A; WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Software Architecture for Shared Information Systems	
12. PERSONAL AUTHOR(S) Mary Shaw	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) March 1993 15. PAGE COUNT 46 pp.
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (continue on reverse of necessary and identify by block number) Allen Newell software architectures info. systems

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Software system design takes place at many levels. Different kinds of design elements, notations, and analyses distinguish these levels. At the *software architecture* level, designers combine subsystems into complete systems. This paper studies some of the common patterns of idioms, that guide these configurations. Results from software architecture offer some insight into the problems of systems integration–the task of connecting individual, isolated, pre-existing software systems to provide coherent, distributed solutions to large problems. As computing has become more sophisticated, so too have the software structures used in the integration task. This paper reviews historical examples of shared information systems in three different applications whose requirements share some common features about collecting, manipulating, and preserving large bodies of complex information. These applications have similar architectural histories in which a succession of designs responds to new technologies and new requirements for flexible, highly dynamic responses. A common pattern, the *shared information systems evolution pattern*, appears in all three

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631 22c. OFFICE SYMBOL ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19

areas.